Control Flow Hijack Defenses
Canaries, DEP, and ASLR

David Brumley
Carnegie Mellon University
Control Flow Hijack: Always control + computation

- code injection
- return-to-libc
- Heap metadata overwrite
- return-oriented programming
- ...

Same principle, different mechanism
Control Flow Hijacks

... happen when an attacker gains control of the instruction pointer.

Two common hijack methods:
- buffer overflows
- format string attacks
Control Flow Hijack Defenses

Bugs are the root cause of hijacks!
• Find bugs with analysis tools
• Prove program correctness

Mitigation Techniques:
• Canaries
• Data Execution Prevention/No eXecute
• Address Space Layout Randomization
## Proposed Defense Scorecard

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>• Smaller impact is better</td>
</tr>
<tr>
<td>Deployment</td>
<td>• Can everyone easily use it?</td>
</tr>
<tr>
<td>Compatibility</td>
<td>• Doesn’t break libraries</td>
</tr>
<tr>
<td>Safety Guarantee</td>
<td>• Completely secure to easy to bypass</td>
</tr>
</tbody>
</table>

Canary / Stack Cookies

```
#include <string.h>

int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:
```
0x080483e4 <+0>:  push   %ebp
0x080483e5 <+1>:  mov    %esp,%ebp
0x080483e7 <+3>:  sub    $72,%esp
0x080483ea <+6>:  mov    12(%ebp),%eax
0x080483ed <+9>:  mov    4(%eax),%eax
0x080483f0 <+12>: mov    %eax,4(%esp)
0x080483f4 <+16>: lea    -64(%ebp),%eax
0x080483f7 <+19>: mov    %eax,(%esp)
0x080483fa <+22>: call   0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```
“A”x68 . “\xEF\xBE\xAD\xDE”

#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
0x080483e4 <+0>: push %ebp
0x080483e5 <+1>: mov %esp,%ebp
0x080483e7 <+3>: sub $72,%esp
0x080483ea <+6>: mov 12(%ebp),%eax
0x080483ed <+9>: mov 4(%eax),%eax
0x080483f0 <+12>: mov %eax,4(%esp)
0x080483f4 <+16>: lea -64(%ebp),%eax
0x080483f7 <+19>: mov %eax,(%esp)
0x080483fa <+22>: call 0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret

Address overwriting:
- argv[1] overwritten with 0xDEADBEEF
- buf overwritten with "AAAA... (64 in total)"
- %esp overwritten
- %ebp overwritten
Idea:

• prologue introduces a *canary word* between return addr and locals

• epilogue checks canary before function returns

Wrong Canary => Overflow
gcc Stack-Smashing Protector (ProPolice)

Dump of assembler code for function main:

```assembly
0x08048440 <+0>: push %ebp
0x08048441 <+1>: mov %esp,%ebp
0x08048443 <+3>: sub $76,%esp
0x08048446 <+6>: mov %gs:20,%eax
0x0804844c <+12>: mov %eax,-4(%ebp)
0x0804844f <+15>: xor %eax,%eax
0x08048451 <+17>: mov 12(%ebp),%eax
0x08048454 <+20>: mov 4(%eax),%eax
0x08048457 <+23>: mov %eax,4(%esp)
0x0804845b <+27>: lea -68(%ebp),%eax
0x0804845e <+30>: mov %eax,(%esp)
0x08048461 <+33>: call 0x8048350 <strcpy@plt>
0x08048466 <+38>: mov -4(%ebp),%edx
0x08048469 <+41>: xor %gs:20,%edx
0x0804846e <+46>: je 0x8048477 <main+55>
0x08048470 <+48>: call 0x8048477 <main+55>
0x08048472 <+50>: call 0x8048340 <__stack_chk_fail@plt>
0x08048477 <+55>: lea
0x08048478 <+56>: ret
```

Compiled with v4.6.1:
```
gcc -fstack-protector -O1 ...
```
Canary should be **HARD** to Forge

- **Terminator Canary**
  - 4 bytes: 0,CR,LF,-1 (low->high)
  - terminate strcpy(), gets(), ...

- **Random Canary**
  - 4 random bytes chosen at load time
  - stored in a guarded page
  - need good randomness
# Canary Scorecard

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Canary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>• several instructions per function</td>
</tr>
<tr>
<td></td>
<td>• time: a few percent on average</td>
</tr>
<tr>
<td></td>
<td>• size: can optimize away in safe functions (but see MS08-067 *)</td>
</tr>
<tr>
<td><strong>Deployment</strong></td>
<td>• recompile suffices; no code change</td>
</tr>
<tr>
<td><strong>Compatibility</strong></td>
<td>• perfect—invisible to outside</td>
</tr>
<tr>
<td><strong>Safety Guarantee</strong></td>
<td>• <em>not really</em>...</td>
</tr>
</tbody>
</table>

Bypass: Data Pointer Subterfuge

Overwrite a data pointer \textit{first}...

```c
int *ptr;
char buf[64];
memcpy(buf, user1);
*ptr = user2;
```
Canary Weakness

Check does **not** happen until epilogue...

- func ptr subterfuge
- C++ vtable hijack
- exception handler hijack
- ...

Code Examples:


VS 2003: /GS
What is “Canary”?  

*Wikipedia*: “the historic practice of using canaries in coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a biological warning system.”
Data Execution Prevention (DEP) / No eXecute (NX)
How to defeat exploits?

shellcode | padding | &buf

\textit{computation} + \textit{control}

DEP

Canary
Data Execution Prevention

Mark stack as non-executable using NX bit

(shellcode padding &buf)

CRASH

(still a Denial-of-Service attack!)
Each memory page is exclusively either writable or executable.

(shellcode padding &buf)

(still a Denial-of-Service attack!)
## DEP Scorecard

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Data Execution Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>• with hardware support: no impact</td>
</tr>
<tr>
<td></td>
<td>• otherwise: reported to be &lt;1% in PaX</td>
</tr>
<tr>
<td><strong>Deployment</strong></td>
<td>• kernel support (common on all platforms)</td>
</tr>
<tr>
<td></td>
<td>• modules opt-in (less frequent in Windows)</td>
</tr>
<tr>
<td><strong>Compatibility</strong></td>
<td>• can break legitimate programs</td>
</tr>
<tr>
<td></td>
<td>- Just-In-Time compilers</td>
</tr>
<tr>
<td></td>
<td>- unpackers</td>
</tr>
<tr>
<td><strong>Safety Guarantee</strong></td>
<td>• code injected to NX pages never execute</td>
</tr>
<tr>
<td></td>
<td>• <em>but code injection may not be necessary</em></td>
</tr>
</tbody>
</table>
# Return-to-libc Attack

Overwrite return address by address of a libc function

- setup fake return address and argument(s)
- `ret` will “call” libc function

No injected code!

<table>
<thead>
<tr>
<th>fake arg 1</th>
<th>fake ret addr</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;system()</code></td>
<td>caller’s ebp</td>
</tr>
</tbody>
</table>

`buf (64 bytes)`
More to come later

return-Oriented
programming
Address Space Layout Randomization (ASLR)

Assigned Reading:
*ASLR Smack and Laugh Reference* by Tilo Muller
Address Space Layout Randomization

Shellcode

Shellcode

Oops...
ASLR

Traditional exploits need precise addresses
  – *stack-based overflows*: location of shell code
  – *return-to-libc*: library addresses

• **Problem**: program’s memory layout is fixed
  – stack, heap, libraries etc.

• **Solution**: randomize addresses of each region!
Running cat Twice

• Run 1

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
082ac000-082cd000  rw-p 082ac000 00:00 0     [heap]
b7dfe000-b7f53000  r-xp 00000000 08:01 1750463 /lib/i686/cmov/libc-2.7.so
b7f53000-b7f54000  r--p 00155000 08:01 1750463 /lib/i686/cmov/libc-2.7.so
b7f54000-b7f56000  rw-p 00156000 08:01 1750463 /lib/i686/cmov/libc-2.7.so
bf966000-bf97b000  rw-p bffeb000 00:00 0     [stack]
```

• Run 2

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
086e8000-08709000  rw-p 086e8000 00:00 0     [heap]
b7d9a000-b7eef000  r-xp 00000000 08:01 1750463 /lib/i686/cmov/libc-2.7.so
b7eef000-b7ef0000  r--p 00155000 08:01 1750463 /lib/i686/cmov/libc-2.7.so
b7ef0000-b7ef2000  rw-p 00156000 08:01 1750463 /lib/i686/cmov/libc-2.7.so
bf902000-bf917000  rw-p bffeb000 00:00 0     [stack]
```
Memory

Program
- Code
- Uninitialized data
- Initialized data

Mapped
- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Stack
- Main stack
ASLR Randomization

\[ a + 16 \text{ bit rand } r_1 \quad b + 16 \text{ bit rand } r_2 \quad c + 24 \text{ bit rand } r_3 \]

- **Program**
  - Code
  - Uninitialized data
  - Initialized data

- **Mapped**
  - Heap
  - Dynamic libraries
  - Thread stacks
  - Shared Memory

- **Stack**
  - Main stack

* \( \approx 16 \text{ bit random number of 32-bit system. More on 64-bit systems.} \)
# ASLR Scorecard

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Address Space Layout Randomization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>• excellent—randomize once at load time</td>
</tr>
<tr>
<td><strong>Deployment</strong></td>
<td>• turn on kernel support <em>(Windows: opt-in per module, but system override exists)</em>&lt;br&gt;• no recompilation necessary</td>
</tr>
<tr>
<td><strong>Compatibility</strong></td>
<td>• transparent to safe apps <em>(position independent)</em></td>
</tr>
<tr>
<td><strong>Safety Guarantee</strong></td>
<td>• not good on x32, much better on x64&lt;br&gt;• <em>code injection may not be necessary</em>...</td>
</tr>
</tbody>
</table>
Ubuntu - ASLR

• **ASLR is ON by default** [Ubuntu-Security]
  – cat /proc/sys/kernel/randomize_va_space
    • Prior to Ubuntu 8.10: **1** (*stack/mmap ASLR*)
    • In later releases: **2** (*stack/mmap/brk ASLR*)
  
  – *stack/mmap ASLR*: since kernel 2.6.15 (Ubuntu 6.06)
  – *brk ASLR*: since kernel 2.6.26 (Ubuntu 8.10)
  – *exec ASLR*: since kernel 2.6.25
    • Position Independent Executable (PIE) with “-fPIE –pie”
How to attack with ASLR?

- Brute Force
- Non-randomized memory
- Stack Juggling
- GOT Hijacking

Attack Methods:
- ret2text
- ret2ret
- ret2pop
- ret2got
- Func ptr
Brute Force

- Shell code
- NOP Sled

brute force search

memory
How to attack with ASLR?

**Brute Force**

- Non-randomized memory
- Stack Juggling
- GOT Hijacking

**Attack**

- ret2text
- ret2ret
- ret2pop
- Func ptr
- ret2got
ret2text

• text section has executable program code
  – but not typically randomized by ASLR except PIE

• can hijack control flow to unintended (but existing) program function
  – Figure 7 in reading
ret2text

Same as running "winner" in vuln2 from class exercise
Function Pointer Subterfuge

Overwrite a function pointer to point to:

- program function (similar to ret2text)
- another lib function in Procedure Linkage Table

```c
/*please call me!*/
int secret(char *input) { ... }

int chk_pwd(char *input) { ... }

int main(int argc, char *argv[]) {
    int (*ptr)(char *input);
    char buf[8];

    ptr = &chk_pwd;
    strncpy(buf, argv[1], 12);
    printf("[] Hello %s!\n", buf);
    (*ptr)(argv[2]);
}
```
Function Pointers

\begin{itemize}
  \item saved ebp
  \item buffer
  \item $\text{x04|x85|x04|x08}$
  \item $\text{AAAAAAAA}$
  \item 4 bytes
  \item 8 bytes
\end{itemize}

\begin{verbatim}
ptr = &chk_pwd;
strncpy(buf, argv[1], 12);
printf("[] Hello %s\n", buf);
(*ptr)(argv[2]);
\end{verbatim}
How to attack with ASLR?

**Attack**

- **Brute Force**
- **Non-randomized memory**
- **Stack Juggling**
- **GOT Hijacking**
  - ret2text
  - Func ptr
  - ret2ret
  - ret2pop
  - ret2got
void \textbf{msglog}(char *\textit{input})\
\{\
    char buf[64];\
    \textbf{strcpy}(buf, \textit{input});\
\}\n
int main(int argc, char *argv[])\
\{\
    if(argc != 2)\
    \{\
        printf("exploitme <msg>\n");\
        return -1;\
    }\
\}

\textbf{msglog}(argv[1]);\
return 0;\
\}

returns pointer to buf in eax

A subsequent call *eax would redirect control to buf
void msglog(char *input) {
    char buf[64];
    strcpy(buf, input);
}

Disassemble to find call *eax
Overwrite with address of call *eax

SHELLCODE

return

saved ebp

4 bytes

64 bytes

ret2eax
ret2ret

• If there is a valuable (potential shellcode) pointer on a stack, you might consider this technique.

```
shellcode (usually resides in buf, but how to point there?)

ret = pop eip; jmp eip;

“stack juggling”
```
ret2ret (stack juggling)

You might consider this technique when
- Text section isn’t randomized (uses addr of ret instr)
- Can overwrite pointer ptr that points to stack
- ptr is higher on the stack than vuln buffer

void f(char *str) {
    char buffer[256];
    strcpy(buffer, str);
}

int main(int argc, char *argv[]) {
    int no = 1;
    int *ptr = &no;
    f(argv[1]);
}
If there is a valuable *(potential shellcode)* pointer on a stack, you might consider this technique.
How to attack with ASLR?

Attack

- Brute Force
- Non-randomized memory
- Stack Juggling
- GOT Hijacking

- ret2text
- ret2ret
- ret2got
- Func ptr
- ret2pop
Other Non-randomized Sections

• Dynamically linked libraries are loaded at runtime. This is called *lazy binding*.

• Two important data structures
  – Global Offset Table
  – Procedure Linkage Table

commonly positioned statically at compile-time
Dynamic Linking

... printf("hello ");
... printf("world\n");
...

<printf@plt>: jmp GOT[printf]

GOT
...
<printf>: dynamic_linker_addr

Transfer control to PLT entry of printf

LIBC
<dynamic_printf_addr>: ...

Linker
Dynamic Linking

printf("hello ");
printf("world\n");
Dynamic Linking

Subsequent calls to printf do not require the linker

```
... printf("hello ");
...
printf("world\n");
...```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_printf_addr
```

```
<dynamic_printf_addr>: ...
```

LIBC

Linker
Exploiting the linking process

• GOT entries are really function pointers positioned at known addresses

• **Idea:** use other vulnerabilities to take control (e.g., format string)
GOT Hijacking

Use the format string to overwrite a GOT entry

LIBC

<dynamic_printf_addr>: ...

Linker

<printf@plt>: jmp GOT[printf]

GOT

...<printf>: dynamic_linker_addr
GOT Hijacking

... printf(usr_input);
... printf("world\n"); ...

Use the format string to overwrite a GOT entry

LIBC
<dynamic_printf_addr>:
...

<printf@plt>: jmp GOT[printf]

GOT
...
<printf>: any_attacker_addr

Linker
GOT Hijacking

```
... printf(usr_input);
...
printf("world\n");
...
```

The next invocation transfers control wherever the attacker wants (e.g., system, pop-ret, etc)

```
<printf@plt>: jmp GOT[printf]
GOT
...
<printf>: any_attacker_addr
```

LIBC
```
<dynamic_printf_addr>: ...
```

Linker
How to attack with ASLR?

**Attack**

- **Brute Force**
- **Non-randomized memory**
- **Stack Juggling**
- **GOT Hijacking**

- **ret2text**
- **ret2ret**
- **ret2got**
- **Func ptr**
- **ret2pop**
Many other techniques

- ret2bss, ret2data, ret2heap, ret2eax
- string pointer
- ret2dtors
  - overwriting dtors section
The Security of ASLR

Optional Reading:

*On the Effectiveness of Address-Space Randomization*

by Shacham et al, ACM CCS 2004
$ /bin/cat /proc/self/maps
08048000-0804f000 r-xp 00000000 08:01 2514948 /bin/cat
0804f000-08050000 rw-p 00006000 08:01 2514948 /bin/cat
08050000-08071000 rw-p 08050000 00:00 0 [heap]
b7d3b000-b7e75000 r--p 00000000 08:01 1475932 /usr/lib/locale/locale-archive
b7e75000-b7e76000 rw-p b7e75000 00:00 0
b7e76000-b7fcb000 r-xp 00000000 08:01 205950 /lib/x86_64-linux-gnu/libc-2.7.so
b7fcb000-b7fcccc000 r--p 00155000 08:01 205950 /lib/x86_64-linux-gnu/libc-2.7.so
b7fcccc000-b7fceed000 rw-p 00156000 08:01 205950 /lib/x86_64-linux-gnu/libc-2.7.so
b7fceed000-b7fd1000 rw-p b7fceed000 00:00 0
b7fe1000-b7fe3000 rw-p b7fe1000 00:00 0
b7fe3000-b7fe4000 r-xp b7fe3000 00:00 0 [vdso]
b7fe4000-b7ffe000 r-xp 00000000 08:01 196610 /lib/ld-2.7.so
b7ffe000-b8000000 rw-p 0001a000 08:01 196610 /lib/ld-2.7.so
bffe0000-c0000000 rw-p bffe0000 00:00 0 [stack]

• ~ 27 bits between bffe0000, bffe0000.
• Top 4 not touched by PAX.
• < ~24 bits of randomness.
• Shacham et al report 16 bits in reality for x86 on Linux.
When to Randomize?

1. When the machine starts? (Windows)
   - Assign each module an address once per boot

2. When a process starts? (Linux)
   - Constant re-randomization for all child processes
Security Game for ASLR

• Attempted attack with randomization guess $x$ is “a probe”
  – Success = $x$ is correct
  – Failure = detectable crash or fail to exploit
  – Assume 16 bits of randomness available for ASLR

• **Game:**
  In expectation, how many probes are necessary to guess $x$?

• *Scenario 1:* not randomized after each probe (Windows)
• *Scenario 2:* re-randomized after each probe (Linux)
What is the expected number of probes to hack the machine?

1. \( \text{Pr}[\text{Success on exactly trial } n]? \)
2. \( \text{Pr}[\text{Success by trial } n]? \)
Scenario 1: Not Randomized After Each Probe

- Pretend that each possible offset is written on a ball.
- There are $2^{16}$ balls.
- This scenario is like selecting balls \textit{without replacement} until we get the ball with the randomization offset written on it.
W/O Replacement:

Pr[Success on Exactly nth try]

Probe 1

\[ \frac{2^{16} - 1}{2^{16}} \]

Success

\[ \frac{1}{2^{16} - 1} \]

Success

Fail, Probe 2

\[ \frac{2^{16} - 1}{2^{16}} \]

\[ \frac{1}{2^{16} - 1} \]

Success

Success

Fail, Probe 3

\[ \frac{2^{16} - 2}{2^{16} - 1} \]

\[ \frac{1}{2^{16} - 2} \]

Success

Success

Fail, Probe 4

\[ \frac{2^{16} - 2}{2^{16} - 3} \]

\[ \frac{1}{2^{16} - 3} \]

Success
W/O Replacement:

Pr[Success on Exactly nth try]

\[
\frac{2^{16} - 1}{2^{16}} \times \frac{2^{16} - 2}{2^{16} - 1} \times \ldots \times \frac{2^{16} - n - 1}{2^{16} - n} \times \frac{1}{2^{16} - n - 1} = \frac{1}{2^{16}}
\]

Fail the first n-1 times

Succeed on nth trial
W/O Replacement:
Pr[Success by nth try] =
Pr[Success on 1st try] +
Pr[Success on 2nd try] +
Pr[Success on nth try] = \frac{n}{2^{16}}
Expected Value

• $E[X]$ is the expected value of random variable $X$
  – Basically a weighted average

\[
E[X] = x_1 p_1 + x_2 p_2 + \ldots + x_k p_k .
\]
\[
E[X] = \sum_{i=1}^{\infty} x_i p_i ,
\]
Expected number of trials before success

$$\text{Pr[success by nth try]}$$

$$\text{Expectation : } \sum_{n=1}^{2^{16}} n \times \frac{1}{2^{16}}$$

$$= \frac{1}{2^{16}} \times \sum_{n=1}^{2^{16}} n$$

$$= \frac{2^{16} + 1}{2}$$
Scenario 2: Randomized After Each Probe

- Pretend that each possible offset is written on a ball.
- There are $2^{16}$ balls.
- Re-randomizing is like selecting balls with replacement until we get the ball with the randomization offset written on it.
With Replacement

\[ \Pr[\text{Success on exactly nth try}] \]

Probe 1
- Success
- \( \frac{2^{16} - 1}{2^{16}} \)

Success
- \( \frac{1}{2^{16}} \)

Fail, Probe 2
- Success
- \( \frac{2^{16} - 1}{2^{16}} \)

Fail, Probe 3
- Success
- \( \frac{1}{2^{16}} \)

Fail, Probe 4
- Success
- \( \frac{1}{2^{16}} \)

Geometric dist.
\[ p = \frac{1}{2^{16}} \]
With Replacement:

Expected number of probes: \( \frac{1}{p} = \frac{1}{2^{16}} \)

\[ E[X] = \frac{1}{p} \text{ for geometric distribution} \]

\[ p = \frac{1}{2^{16}} \]
Comparison

Expected success in $2^{16}$ probes

With Re-Randomization

For $n$ bits of randomness: $2^n$

Without Re-Randomization

For $n$ bits of randomness: $2^{n-1}$

Re-Randomization gives (only) 1 bit of extra security!
But wait...

That’s true, but is brute force the only attack?
Questions?
Backup slides here.

- Titled cherries because they are for the pickin. (credit due to maverick for wit)
Last Two Lectures

Control flow hijacks are due to BUGS!
Format String Attacks

Microsoft took a drastic measure:
%n is disabled by default

• since VS 2005

• int _set_printf_count_output(
  • int enable
• );